SHOVEL: A SAT-based Tool for Information Flow Alarm Classification

Jong-Gwon Kim	Woosuk Lee	Jaeseung Choi	Chung-Kil Hur	Kwangkeun Yi
Seoul Nat'l Univ.	*Georgia Tech*	*Seoul Nat'l Univ.*	*Seoul Nat'l Univ.*	*Seoul Nat'l Univ.*
Korea	*USA*	*Korea*	*Korea*	*Korea*
jgkim@ropas.snu.ac.kr	*woosuk.lee@cc.gatech.edu*	*jschoi@ropas.snu.ac.kr*	*gil.hur@sf.snu.ac.kr*	*kwang@ropas.snu.ac.kr*

***Abstract*—We present a tool, called SHOVEL, that assists the user to quickly classify, as true or false, alarms reported by information flow analyzers. Specifically, SHOVEL helps the user by finding a shortest function call-return path from the source to the sink that satisfies a given user constraint. In this paper, we empirically show that our approach is very effective by classifying 351 alarms for 42 open-source C programs and identifying 48 true alarms with crash bugs, three of which were assigned CVE numbers. We also report the patterns of using SHOVEL during our manual alarm classification.**

SHOVEL finds a shortest path using an off-the-shelf MaxSAT solver taking as user constraint a Boolean formula on the call/return edges of the call graph. For this, we develop a novel algorithm that encodes the constrained shortest path problem into a Boolean formula in a counter-example guided refinement fashion. Our algorithm is very easy to implement but also very efficient. In our empirical study, SHOVEL responded mostly within a few seconds even for programs of 100K LOC.

***Keywords*-static analysis; software reliability; error diagnosis; maximum satisfiability; alarm classification**

I. INTRODUCTION

One of the main practical problems with using static analyzers is that identifying false alarms is highly time-consuming. Since static analysis problems are mostly undecidable, all static analyzers produce (often many) false alarms. To classify alarms as true or false, one may sometimes have to carefully examine the whole source code, which is very tedious and time-consuming. This high cost of manual alarm classification has been one of the major reasons for the underuse of static analysis tools [21], [24].

In this paper, we present a tool, called SHOVEL, that can greatly help the user to determine which alarms are true for information flow analyses. We assume a static analyzer generates the call graph of an input program and reports a set of node pairs in the graph. Each pair of nodes, called *source* and *sink*, is an alarm that information of our interest may flow between the two nodes (*e.g.,* taint flow [16], resource leak [22], privacy leak [23]). In this setting, SHOVEL efficiently finds a shortest well-formed call-return path from the source to the sink that satisfies a user constraint given as a Boolean formula on the call/return edges of the graph.

The reason why SHOVEL finds paths in the abstract domain of call graphs with Boolean constraints rather than in a more concrete domain, for example, of control-flow graphs with constraints in more expressive logic is twofold. First, since the size of a call graph is much smaller than that of a control-flow graph, SHOVEL can scale well to large programs. Indeed, SHOVEL responds mostly within a few seconds even for C programs of 100K LOC. Second, since a user constraint is given as a Boolean formula, SHOVEL can use an off-the-shelf MaxSAT [5] solver to find a shortest path satisfying the constraint. We use a MaxSAT sovler to find a minimal solution of a Boolean formula in terms of the number of the variables assigned true.

Algorithm 1 User Interaction Loop

Global Input G: call graph, f_{src}: source, f_{snk}: sink

Function ClassifyAlarm()

output YES: true alarm, NO: false alarm

1: $\mu := \texttt{true}$ // Initialize
2: **loop**
3: $p := \text{SHOVEL}(\mu)$ // Find a shortest path
4: **if** $p = \text{UNSAT}$ **then return** NO
5: $\delta := \text{UserConstraint}(p)$ // Find a refinement
6: **if** $\delta = \texttt{true}$ **then return** YES
7: $\mu := \mu \wedge \delta$ // Refine
8: **end loop**

Moreover, we empirically show that call graph paths and Boolean constraints are not too abstract to be used: we manually classify alarms using SHOVEL to find many new bugs in various open-source C programs. We applied SHOVEL to two kinds of static analysis problems. One is to check whether tainted user information flows into a format string argument of library functions such as `printf`. The other is to check whether tainted user information with integer overflow flows into the size argument of `malloc`. We used SHOVEL to classify 351 alarms for 42 open-source C programs and identified 48 true alarms with crash bugs, three of which were assigned CVE numbers [10], [11], [12].

Throughout the paper, we present (i) how we manually identify true/false alarms using SHOVEL and (ii) how we develop SHOVEL using a MaxSAT solver.

A. Manual Alarm Classification using SHOVEL

We interactively use SHOVEL to decide whether a given

Algorithm 2 Main Loop of SHOVEL

Global Input G: call graph, f_{src}: source, f_{snk}: sink

Function SHOVEL(μ)
input User constraint μ
output UNSAT or a shortest path satisfying μ

1: $\varphi := \mu \wedge \mathsf{PathEncoding}(\mu)$ // Initialize
2: **loop**
3: $p := \mathsf{FindMinSAT}(\varphi)$ // Solve
4: **if** $p = \text{UNSAT}$ or $\mathsf{WellFormed}(p)$ **then return** p
5: $\varphi := \varphi \wedge \mathsf{PathRefine}(p)$ // Refine
6: **end loop**

alarm is true or not as in Algorithm 1. An alarm is given as a source-sink pair of nodes $(f_{\text{src}}, f_{\text{snk}})$ in the call graph G and we decide whether there is tainted information flow (henceforce called simply *taint flow*) from f_{src} to f_{snk} in the original program. First, we set the user constraint μ to be `true` (*i.e.,* no constraint) (line 1) and find a shortest well-formed path p from f_{src} to f_{snk} satisfying μ using SHOVEL (line 3). When there is no such path, we conclude that the given alarm is false (line 4). Otherwise we examine the source code to decide whether there is taint flow along the path p (line 5). In case of yes, we discover a true bug and stop (line 6). Otherwise, we find a reason why p is spurious, from which we derive a general condition δ that excludes p but includes all (or most) valid paths (*i.e.,* with taint flow) (line 5). Then we refine the user constraint μ with δ (line 7) and repeat the process until we succeed to classify the alarm.

Through our empirical study, we show that our approach is practical making the following observations.

- It is not hard for the user to recover the concrete control flow from a given rather abstract call-return path. The reason is because the user just needs to figure out the control flow inside each function, which is not so hard since the function size is usually not too big.
- Boolean logic is expressive enough to describe our user constraints. In Section III, we give the patterns of the user constraints we have used in detail.
- We could derive general (and sound) constraints based on our intuition and understanding about the program. Inferring such general constraints is the most important role of the user, which can be hardly done by automatic tools.

B. Algorithm of SHOVEL using a MaxSAT Solver

Finding a shortest path satisfying a Boolean constraint is in general at least NP-complete in the size of the constraint. The reason is simple. Consider the constraint that the path should visit all the functions in the call graph with only function calls (*i.e.,* no returns). This constraint can be easily encoded as a Boolean formula whose size is proportional to that of the graph. Then the problem of finding a shortest path

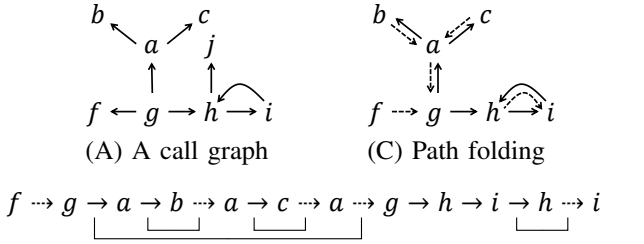


Figure 1. Examples of call graph and well-formed path

under the constraint becomes the Hamiltonian path problem, which is NP-complete in the size of the graph.

To solve this hard problem, we develop a novel algorithm using a MaxSAT solver. The idea is to represent edges of the call graph as Boolean variables and encode the condition for well-formed paths as a Boolean formula on the variables. Then we can find a shortest path by finding a minimal solution of the formula using a MaxSAT solver.

More precisely, since it is hard to exactly encode the path condition, we use the so-called counter example guided refinement approach using sound but not complete encodings (*i.e.,* including all well-formed paths but not excluding all ill-formed ones) as in Algorithm 2. We initialize the Boolean formula φ with the user constraint μ and our initial encoding $\mathsf{PathEncoding}(\mu)$ of well-formed paths (line 1). Then we find a minimal solution p of the formula φ using a MaxSAT solver (line 3). Since our encoding is sound, if φ is unsatisfiable or p is a well-formed path, then we have a correct answer and thus return p (line 4). However, since our encoding is not complete, the minimal solution p may be an ill-formed path. In that case, we update our encoding φ with our sound refinement $\mathsf{PathRefine}(p)$ guided by the counter example p (line 5) and repeat the process until we succeed to find a correct answer.

C. Our Contribution

- We develop a novel algorithm for soundly and efficiently finding a shortest call-return path satisfying a Boolean constraint using an off-the-shelf MaxSAT solver. Our soundness proof is available at the project website.
- Using SHOVEL and the static analyzer SPARROW [30], [26], we found 48 crash bugs, three of which were assigned CVE numbers [10], [11], [12]
- We summarize the user constraint patterns that we have used in our experiment.

II. OVERVIEW

We present the main ideas of the paper using concrete examples.

A. Boolean Representation using Backbone-Branch Decomposition

We first illustrate how we represent a well-formed path as an assignment to Boolean variables.

Consider the call graph in Figure 1.(A), where the nodes and edges represent functions and possible calls between them. The graph also implicitly implies that there is a return edge $w \dashrightarrow v$ for each call edge $v \rightarrow w$. Then consider the well-formed path from f to i depicted in Figure 1.(B), where the solid and dashed arrows represent function calls and returns respectively.

This path is well-formed because every pair of corresponding call-return edges is well-matched in the sense that they are in the opposite direction between the same pair of functions. Specifically, in Figure 1.(B), the nested underlines denote all corresponding call-return pairs in the path, each of which is in the opposite direction. This notion of well-formedness captures the property that every invoked function should return to its caller. An example of ill-formed path is $g \rightarrow h \dashrightarrow i$, where the corresponding call $g \rightarrow h$ and return $h \dashrightarrow i$ are not in the opposite direction (*i.e.,* the invoked function h does not return to its caller g).

The first problem with Boolean encoding of such well-formed paths is to soundly and efficiently encode the well-formedness condition. Since, as we just have seen, corresponding call-return pairs can be nested (*i.e.,* defined by a context-free grammar), it is not obvious how to express such a property in Boolean logic.

Our solution to the problem is to obtain well-formedness by construction via what we call *backbone-branch decomposition*.

Backbone-Branch Decomposition The decomposition of a well-formed path is to classify every edge in the path into two categories: the edges in all corresponding call-return pairs, called *branches*, and those in the rest, called *backbone*. This decomposition can be well illustrated by folding all call-return pairs in the branches. For instance, the example path can be folded as in Figure 1.(C), where $f \dashrightarrow g \rightarrow h \rightarrow i$ is the backbone and the others are branches.

The backbone can be further decomposed into two: the return backbone consisting of return edges, followed by the call backbone consisting of call edges. For example, in Figure 1.(C), the backbone is decomposed into $f \dashrightarrow g$ followed by $g \rightarrow h \rightarrow i$. Such decomposition is possible because there cannot be any call edge immediately followed by a return edge in the backbone, which would form a branch if any.

Boolean Representation With the decomposition, we can obtain well-formedness for free by separately representing the three components (*i.e.,* return/call backbones and branches). Specifically, we represent a well-formed path using three Boolean variables $x^{\mathtt{r}}_{w,v}, x^{\mathtt{c}}_{v,w}, x^{\mathtt{b}}_{v,w}$ for each edge $v \rightarrow w$ in the call graph. The idea is that the variable $x^{\mathtt{r}}_{w,v}$ indicates whether the return edge $w \dashrightarrow v$ is included in the return backbone of the path; $x^{\mathtt{c}}_{v,w}$ whether $v \rightarrow w$ in the call backbone; and $x^{\mathtt{b}}_{v,w}$ whether the pair $v \leftrightarrows w$ in the branches. For example, the well-formed path in Figure 1 is represented by assigning truth values to the following variables:

$$x^{\mathtt{r}}_{f,g}, x^{\mathtt{c}}_{g,h}, x^{\mathtt{c}}_{h,i}, x^{\mathtt{b}}_{g,a}, x^{\mathtt{b}}_{a,b}, x^{\mathtt{b}}_{a,c}, x^{\mathtt{b}}_{i,h} \ .$$

Note that we can avoid encoding well-formedness by using a single branch variable for a call-return pair. Also, note that we encode the image of a call-return path rather than the path itself. For example, two different call-return paths $f \rightarrow g \dashrightarrow f \rightarrow h \dashrightarrow f$ and $f \rightarrow h \dashrightarrow f \rightarrow g \dashrightarrow f$ have the same Boolean representation. However, this abstraction is not problematic for the user to recover a concrete control flow.

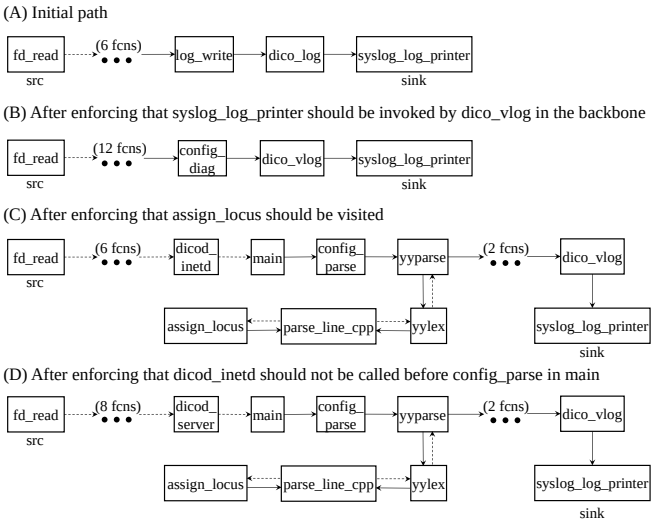


Figure 2. Shortest paths in GNU dicod-2.0 found by SHOVEL

B. Manual Alarm Classification using SHOVEL

Before we present how we develop SHOVEL, we first illustrate how we use SHOVEL in alarm classification.

To give a high-level idea, we present how we classified a format string vulnerability alarm for GNU dicod-2.0. The alarm consists of the source function `fd_read`, which takes tainted user input via the system call `read`, and the sink function `syslog_log_printer`, which passes a format string to the library function `vsnprintf`. Our goal is to determine whether the tainted information from `read` flows into the format string argument of `vsnprintf` in an insecure way.

Step 1 To this end, we first obtained a shortest path using SHOVEL, which is depicted in Figure 2.(A). By examining information flow along this path, we found that `log_write` passes the constant string `"%.*s"` to `dico_log` as shown below, which blocks taint flow.

```
log_write(...){
  ...
  dico_log(p->level, 0, "%.*s", size, buf);
  ...
}
```

Based on this observation, we derived a general constraint that excludes the path as follows. First we checked all the call sites of `dico_log` using the source code browsing tool CSCOPE and found that a constant string is always passed to `dico_log`, which blocks tainted information flow. Thus we can derive the constraint that taint propagation paths should not contain the call from `dico_log` to `syslog_log_printer` in the backbone.

Then we further generalized the constraint as follows. By examining all the call sites of `syslog_log_printer`, we found out that all the callers except `dico_vlog` passes a constant string to `syslog_log_printer` in a similar way as in `dico_log`. From this we can derive the constraint that taint propagation paths should contain the call from `dico_vlog` to `syslog_log_printer` in the backbone.

Step 2 With this constraint, we obtained the shortest path given in Figure 2.(B) using SHOVEL. By examining the path using CSCOPE, we made the following observations. First, `config_diag` is the only caller of `dico_vlog`. Second, in `config_diag`, tainted information may flow from the argument `fmt` and the structure field `locus->file` to the function `dico_vlog`, as shown below.

```
config_diag(..., const char *fmt, ...){
  ...
  asprintf(&newfmt, "%s:%d:warning: %s",
        locus->file, locus->line, fmt);
  ...
  dico_vlog(category, errcode, newfmt, ap);
  ...
}
```

Third, a constant string is passed to `config_diag` for the `fmt` argument at all its call sites, and thus `locus->file` is the only source of taint since `locus->line` is of integer type. Finally, `assign_locus` is the only function that stores a value to a structure field with name `file`.

From these observations we can easily derive the constraint that taint propagation paths should visit the function `assign_locus`. This constraint can be encoded by taking the disjunction of all Boolean variables for incoming and outgoing edges of `assign_locus`.

Step 3 With this additional constraint, we obtained the shortest path given in Figure 2.(C) using SHOVEL. By examining the path using CSCOPE, we observed that the path is infeasible because, in the `main` function, the call to `config_parse` comes before the call to `dicod_inetd`. From this we can derive the constraint that the backbone should not include both the return from `dicod_inetd` to `main` and the call from `main` to `config_parse` at the same time.

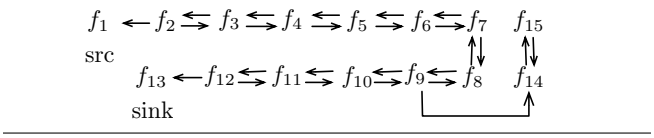


Figure 3. A running example for Boolean encoding

Step 4 With the additional constraint, we obtained the shortest path given in Figure 2.(D) using SHOVEL. Similarly as in Step 3, `dicod_server` cannot be called before `config_parse` in the `main` function and thus we have the constraint that the backbone should not include both the return from `dicod_server` to `main` and the call from `main` to `config_parse` at the same time.

Step 5 With this additional constraint, SHOVEL finally answers that there is no such path satisfying all the constraints given so far. Therefore we can conclude that the alarm from `fd_read` to `syslog_log_printer` is false.

C. Boolean Encoding of Well-formed Paths

We now illustrate how to soundly encode and refine well-formed paths as a Boolean formula (*i.e.,* PathEncoding and PathRefine of Algorithm 2) using the running example of a call graph given in Figure 3.

Our key idea is to encode a certain property $\Phi(V)$ for a set of functions V that every well-formed path should satisfy when passing through V, and apply Φ to carefully chosen sets of functions.

Encoding of Φ The property $\Phi(V)$ is straightforward and easy to encode. For example, for $V = \{ f_5 \}$ in the call graph in Figure 3, one of the sub-properties of $\Phi(V)$ says that if the return backbone of any well-formed path enters V, then its return or call backbone should exit V. This can be easily encoded as follows:

$$x^{\mathtt{r}}_{4,5} \vee x^{\mathtt{r}}_{6,5} \implies x^{\mathtt{r}}_{5,4} \vee x^{\mathtt{r}}_{5,6} \vee x^{\mathtt{c}}_{5,4} \vee x^{\mathtt{c}}_{5,6} \ .$$

The full definition of $\Phi(V)$ is given in Figure 4. The formula $\mathrm{Init}(V)$ states that when V includes the source but not the sink, the return or call backbone should exit V (see (1)); and similarly for the opposite case (see (2)). The formula $\mathrm{IO}(V)$ states that when V does not include the sink, if the return backbone enters V, the return or call backbone should exit V and if the call backbone enters V, the call backbone should exit V (see (3)). The formula $\mathrm{OI}(V)$ states similarly for the opposite case (see (4)). The formula $\mathrm{BR}(V)$ states that when V includes neither the source nor the sink, if the branches exit V, then either the branches enter V or the backbone should pass through V (see (5)).

It is not hard to see that $\Phi(V)$ is sound[1] (*i.e.,* contains every well-formed path from f_{src} to f_{snk}) for an arbitrary

[1]The proof of soundness is available at the project website.

$$\Phi(V) = \mathrm{Init}(V) \wedge \mathrm{IO}(V) \wedge \mathrm{OI}(V) \wedge \mathrm{BR}(V)$$

$$\mathrm{Init}(V) = \begin{cases} \overset{\bullet\!\!\to\!\!\circ}{\mathrm{R}}(V) \vee \overset{\bullet\!\!\to\!\!\circ}{\mathrm{C}}(V) & \text{if } f_{\mathrm{src}} \in V, f_{\mathrm{snk}} \notin V \quad (1) \\ \overset{\circ\!\!\to\!\!\bullet}{\mathrm{R}}(V) \vee \overset{\circ\!\!\to\!\!\bullet}{\mathrm{C}}(V) & \text{if } f_{\mathrm{src}} \notin V, f_{\mathrm{snk}} \in V \quad (2) \\ \mathtt{true} & \text{otherwise} \end{cases}$$

$$\mathrm{IO}(V) = \begin{cases} (\overset{\bullet\!\!\to\!\!\bullet}{\mathrm{R}}(V) \Rightarrow \overset{\bullet\!\!\to\!\!\circ}{\mathrm{R}}(V) \vee \overset{\bullet\!\!\to\!\!\circ}{\mathrm{C}}(V)) \wedge & \text{if } f_{\mathrm{snk}} \notin V \quad (3) \\ \quad (\overset{\bullet\!\!\to\!\!\bullet}{\mathrm{C}}(V) \Rightarrow \overset{\bullet\!\!\to\!\!\circ}{\mathrm{C}}(V)) & \\ \mathtt{true} & \text{otherwise} \end{cases}$$

$$\mathrm{OI}(V) = \begin{cases} (\overset{\bullet\!\!\to\!\!\bullet}{\mathrm{R}}(V) \Rightarrow \overset{\circ\!\!\to\!\!\bullet}{\mathrm{R}}(V)) \wedge & \text{if } f_{\mathrm{src}} \notin V \quad (4) \\ \quad (\overset{\bullet\!\!\to\!\!\bullet}{\mathrm{C}}(V) \Rightarrow \overset{\circ\!\!\to\!\!\bullet}{\mathrm{R}}(V) \vee \overset{\circ\!\!\to\!\!\bullet}{\mathrm{C}}(V)) & \\ \mathtt{true} & \text{otherwise} \end{cases}$$

$$\mathrm{BR}(V) = \begin{cases} \overset{\bullet\!\!\to\!\!\bullet}{\mathrm{B}}(V) \Rightarrow \overset{\circ\!\!\to\!\!\bullet}{\mathrm{B}}(V) \vee \mathrm{RC}(V) & \text{if } f_{\mathrm{src}}, f_{\mathrm{snk}} \notin V \quad (5) \\ \mathtt{true} & \text{otherwise} \end{cases}$$

where $\mathrm{RC}(V) = \overset{\bullet\!\!\to\!\!\bullet}{\mathrm{R}}(V) \vee \overset{\bullet\!\!\to\!\!\bullet}{\mathrm{R}}(V) \vee \overset{\bullet\!\!\to\!\!\bullet}{\mathrm{C}}(V) \vee \overset{\bullet\!\!\to\!\!\bullet}{\mathrm{C}}(V)$

For $(L, l) \in \{ (R, \mathtt{r}), (C, \mathtt{c}), (B, \mathtt{b}) \}$,

$$\begin{aligned} \overset{\bullet\!\!\to\!\!\bullet}{\mathrm{L}}(V) &= \bigvee \{ x^l_{v,w} \mid v \in V \} \\ \overset{\bullet\!\!\to\!\!\bullet}{\mathrm{L}}(V) &= \bigvee \{ x^l_{v,w} \mid w \in V \} \\ \overset{\bullet\!\!\to\!\!\circ}{\mathrm{L}}(V) &= \bigvee \{ x^l_{v,w} \mid v \in V, w \notin V \} \\ \overset{\circ\!\!\to\!\!\bullet}{\mathrm{L}}(V) &= \bigvee \{ x^l_{v,w} \mid v \notin V, w \in V \} \end{aligned}$$

Figure 4. Definition of $\Phi(V)$ for source f_{src} and sink f_{snk}.

set V of functions because the properties we encode are straightforward. Thus the conjunction of $\Phi(V)$'s for arbitrary choices of V is also sound.

Application of Φ We now define $\mathsf{PathEncoding}(\mu)$ and $\mathsf{PathRefine}(p)$ of Algorithm 2 by applying Φ to well chosen sets of functions:

$$\begin{aligned} \mathsf{PathEncoding}(\mu) &= \bigwedge \{ \Phi(V) \mid V \in \mathsf{InitGrps}(\mu) \} \\ \mathsf{PathRefine}(p) &= \bigwedge \{ \Phi(V) \mid V \in \mathsf{RefiGrps}(p) \} \wedge \mathsf{Not}(p) \end{aligned}$$

where $\mathsf{Not}(p)$ is the Boolean formula that negates exactly the solution p. We add $\mathsf{Not}(p)$ in order to formally guarantee that $\mathsf{PathRefine}(p)$ excludes p.

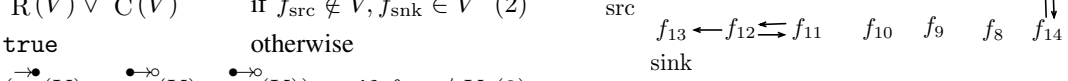
Then we illustrate how to find good collections of functions for $\mathsf{InitGrps}$ and $\mathsf{RefiGrps}$ with a running example. Consider the call graph in Figure 3 with f_1 source and f_{13} sink together with the user constraint μ given by

$$\mu = \neg x^{\mathtt{r}}_{2,3} \wedge \neg (x^{\mathtt{c}}_{10,9} \vee x^{\mathtt{b}}_{10,9}) \wedge (x^{\mathtt{c}}_{15,14} \vee x^{\mathtt{b}}_{15,14}) \ .$$

Basic Initial Groupings Our basic idea is to apply Φ to every singleton set in the call graph G. In the running example, we have the following groupings:

$$\mathsf{InitGrps}(\mu) = \{ \{f_1\}, \ldots, \{f_{15}\} \}$$

In this case, a minimal solution of $\mu \wedge \mathsf{PathEncoding}(\mu)$ with size 8 is given as follows:



(CE1)

The main reason for the solution being ill-formed is that the path can be disconnected by making cycles.

Advanced Initial Groupings In order to alleviate the disconnection problem, we apply Φ to additional function sets derived by means of logical inference from μ and SCC (Strongly Connected Component) computation. The details are as follows.

First, we find return edges $v \dashrightarrow w$ such that their absence in the return backbone is derivable from the user constraint μ (*i.e.,* $\mu \Rightarrow \neg x^{\mathtt{r}}_{v,w}$ provable). Then we compute all SCCs of the sub-graph of the original call graph that consists of only and all the return edges except those we just derived from μ. Then we apply Φ to those SCCs. In the running example, since we can derive $\neg x^{\mathtt{r}}_{2,3}$ from μ, we compute the SCCs from the return edges of the call graph in Figure 3 except $f_2 \dashrightarrow f_3$, which results in the following (non-singleton) groupings:

$$\{f_3, \ldots, f_{12}\}, \{f_{14}, f_{15}\}$$

Second, we perform a similar process for call backbone variables. We find call edges $v \to w$ such that $\mu \Rightarrow \neg x^{\mathtt{c}}_{v,w}$ is provable and then compute the SCCs from the call edges of the call graph except those just found. In the running example, we can derive $\neg x^{\mathtt{c}}_{10,9}$ from μ and obtain the following (non-singleton) SCCs:

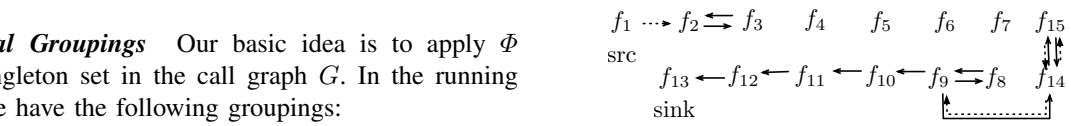
$$\{f_2, \ldots, f_8\}, \{f_9, \ldots, f_{12}\}, \{f_{14}, f_{15}\}$$

Finally, we perform a similar process for branch variables. We find call edges $v \to w$ such that $\mu \Rightarrow \neg x^{\mathtt{b}}_{v,w}$ is provable and conduct the same process as before. In the running example, we can derive $\neg x^{\mathtt{b}}_{10,9}$, which results in the same SCCs as for the call backbone.

By taking all the groupings together, we have:

$$\begin{aligned} \mathsf{InitGrps}(\mu) = \{ &\{f_1\}, \ldots, \{f_{15}\}, \{f_3, \ldots, f_{12}\}, \{f_{14}, f_{15}\}, \\ &\{f_2, \ldots, f_8\}, \{f_9, \ldots, f_{12}\} \} \end{aligned}$$

This time a minimal solution of $\mu \wedge \mathsf{PathEncoding}(\mu)$ with size 12 is given as follows:



(CE2)

This path is still ill-formed but better than the counter example (CE1) obtained by the basic groupings. Also note that the path (CE1) does not satisfy this advanced initial formula

$\Phi(\mathsf{InitGrps}(\mu))$ because, for example, the path enters but does not exit the group $\{f_2, \ldots, f_8\}$.

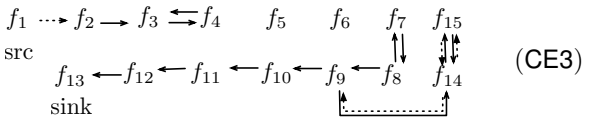
Basic Refinement Groupings Now we show how to refine the initial groupings using a counter example. The basic idea is to simply apply Φ to the SCCs computed from the counter example. More specifically, we compute SCCs separately for the return backbone, the call backbone and the branches. For the counter example (CE2) above, we have the following non-singleton SCCs:

$$\begin{aligned} \text{Return backbone}: &\quad \text{None} \\ \text{Call backbone}: &\quad \{f_2, f_3\}, \{f_8, f_9\} \\ \text{Branches}: &\quad \{f_{14}, f_{15}\} \end{aligned}$$

Thus to the initial groupings, we add

$$\mathsf{RefiGrps}(\mathsf{CE2}) = \{\, \{f_2, f_3\}, \{f_8, f_9\}, \{f_{14}, f_{15}\} \,\}$$

and find a minimal solution using a MaxSAT solver, which is of size 14 and given as follows:

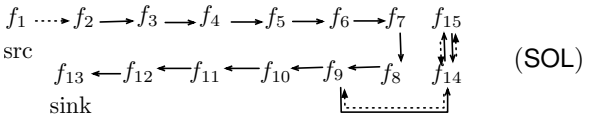


(CE3)

By applying the same refinement process again to the counter example (CE3), we have

$$\mathsf{RefiGrps}(\mathsf{CE3}) = \{\, \{f_3, f_4\}, \{f_7, f_8\}, \{f_{14}, f_{15}\} \,\}$$

and the following minimal solution of size 15:



(SOL)

This path is finally well-formed and thus a shortest path satisfying the user constraint μ.

Advanced Refinement Groupings We can improve the basic refinement groupings to make bigger progress in a single refinement step. The idea is to take larger groups than the SCCs of the counter example by including their neighbor functions. To maximize efficiency, however, we have to carefully select neighbors. The details of our selection scheme is given in Appendix VII.

Here we illustrate the high-level idea using the running example. Recall that for the counter example (CE2) we obtain the groupings $\{f_2, f_3\}, \{f_8, f_9\}, \{f_{14}, f_{15}\}$ by computing SCCs. Now we enlarge the groupings by adding those neighbors that are chosen by our selection scheme. The result is as follows:

$$\begin{aligned} \mathsf{RefiGrps}(\mathsf{CE2}) = \{\, &\{f_2, f_3\}, \{f_8, f_9\}, \{f_{14}, f_{15}\}, \\ &\{f_2, f_3, f_4\}, \{f_7, f_8, f_9\} \,\} \end{aligned}$$

where we add to $\{f_2, f_3\}$ one of its neighbors, f_4, and to $\{f_8, f_9\}$ one of its neighbors, f_7.

With this advanced refinement groupings, we directly find the well-formed path (SOL) from the counter example (CE2) in one step. Note that the path (CE3) does not satisfy this advanced refinement formula $\Phi(\mathsf{RefiGrps}(\mathsf{CE2}))$ because, for example, the path enters but does not exit the group $\{f_2, f_3, f_4\}$.

III. User Constraint Patterns

In this section, we summarize the user constraint patterns that we have used in our experiment.

First of all, our patterns apply around a function, we say f throughout the section, and are sound only when every taint flow from the source to the sink should pass through f. Though one may think this condition may seriously restrict the applicability of our patterns, it turns out that it was not problematic most of the time in our experiment. The reason is because we apply the patterns to functions near the source or the sink, which are likely to satisfy the condition.

However, we also have a work-around even when the condition fails, which occasionally happened during our experiment. The idea is to first find a Boolean condition ρ such that every taint flow satisfying ρ passes through f; then put the user constraint μ derived by our patterns under the premise ρ (*i.e.,* $\rho \Rightarrow \mu$).

It is important to note that the patterns presented here may not be sound in a certain situation though they are sound in most cases. Thus before applying one of the patterns, the user has to check whether it is (likely to be) sound under the current situation, which is usually easy in our experience.

A. Examining functions that f calls

This pattern is to examine functions invoked inside f.

```
f(..) {.. g1(..); .. g2(..); .. g3(..); ..}
```

By examining the code of f we observe either (i) that taints may flow from argument values of f (or the source point inside f) to only particular functions (suppose g_1, g_2, g_3 here) typically when f is near the source; or (ii) that taints may flow from only particular functions (suppose g_1, g_2, g_3 here) to return values of f (or the sink point inside f) typically when f is near the sink. Then by examining each function (here g_1, g_2, g_3), we determine whether taint flow gets blocked in it. For example, suppose we discover that g_2 blocks taint flow.

Finally, we give the constraint, in case of (i), that in the backbone there should be a call from f to one of the functions that do not block taint flow; and in case of (ii) a return to f from one of those functions. In the example, the constraint is given by $x^{\mathsf{c}}_{f,g_1} \vee x^{\mathsf{c}}_{f,g_3}$ in case of (i) and $x^{\mathsf{r}}_{g_1,f} \vee x^{\mathsf{r}}_{g_3,f}$ in case of (ii).

B. Examining functions that calls f

This pattern is to examine call sites of f. For example, suppose using a source browsing tool we find that only g_1,

g_2, g_3 invoke f.

```
g1(..) { .. f(..); .. }
g2(..) { .. f(..); .. }
g3(..) { .. f(..); .. }
```

Then by examining each function (here g_1, g_2, g_3), we determine either (i) whether taints may flow from the function's argument values to f typically when f is near the sink; or (ii) whether taints may flow from f to the function's return values typically when f is near the source. For example, suppose we discover that taint flow is blocked in g_2.

Finally, we give the constraint, in case of (i), that in the backbone there should be a call to f from one of the functions that do not block taint flow; and in case of (ii) a return from f to one of those functions. In the example, the constraint is given by $x^{\mathtt{c}}_{g_1,f} \vee x^{\mathtt{c}}_{g_3,f}$ in case of (i) and $x^{\mathtt{r}}_{f,g_1} \vee x^{\mathtt{r}}_{f,g_3}$ in case of (ii).

C. Examining a call/return chain along the path

This pattern is to examine a call/return chain along the given shortest path and exclude the chain from the backbone if it blocks taint flow (*e.g.,* by sanitizing the taint). Specifically we have four cases:
(i) a call chain starting from f (*e.g.,* $f \rightarrow g_1 \rightarrow g_2$);
(ii) a return chain ending in f (*e.g.,* g_2 -$\!\!\rightarrow g_1$ -$\!\!\rightarrow f$);

```
 f(..) { .. g1(..); .. }
g1(..) { .. g2(..); .. }
```

(iii) a call chain ending in f (*e.g.,* $g_1 \rightarrow g_2 \rightarrow f$);
(iv) a return chain starting from f (*e.g.,* f -$\!\!\rightarrow g_2$ -$\!\!\rightarrow g_1$).

```
g1(..) { .. g2(..); .. }
g2(..) { ..  f(..); .. }
```

Boolean encoding of the constraints excluding these call chains is easy. For example, the four example chains can be excluded by $\neg(x^{\mathtt{c}}_{f,g_1} \wedge x^{\mathtt{c}}_{g_1,g_2})$, $\neg(x^{\mathtt{r}}_{g_2,g_1} \wedge x^{\mathtt{r}}_{g_1,f})$, $\neg(x^{\mathtt{c}}_{g_1,g_2} \wedge x^{\mathtt{c}}_{g_2,f})$, and $\neg(x^{\mathtt{r}}_{f,g_2} \wedge x^{\mathtt{r}}_{g_2,g_1})$, respectively.

D. Examining functions that update a particular variable

This pattern is to find all the places where a particular global variable or structure field is updated. We usually find such places using a source browsing tool by searching for the name of the variable or field. We use this pattern when we observe by examining f that taints should be read from a particular variable/field and thus taint flow should pass through one of the places where the variable/field is updated.

```
 f(..) { .. x = name; .. }
g1(..) { .. name = y; .. }
g2(..) { .. name = z; .. }
```

or

```
 f(..) { .. x = a->name; .. }
g1(..) { .. b->name = y; .. }
g2(..) { .. c->name = z; .. }
```

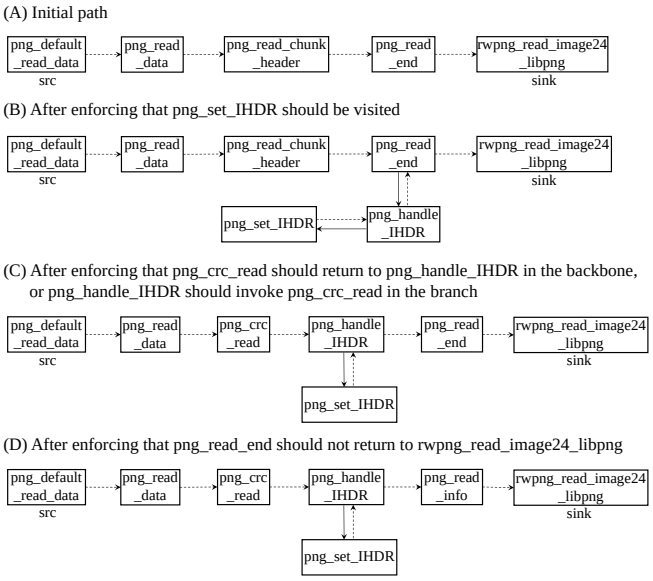


Figure 5. Shortest paths in pngquant-2.7.0 found by SHOVEL

For example, suppose we found that the global variable or structure field `name` is updated only in g_1 and g_2. Then we can easily encode the constraint that either g_1 or g_2 should be visited by taking the disjunction of all Boolean variables of the three type (*i.e.,* $x^{\mathtt{r}}, x^{\mathtt{c}}, x^{\mathtt{b}}$) for all incoming and outgoing edges of g_1 and g_2.

E. Examining the order between two function calls

This pattern is to exclude an infeasible function call order. For example, consider the following code.

```
f(..) { .. g1(..); .. g2(..); .. }
```

Suppose the backbone of the given shortest path constraints the chain g_2 -$\!\!\rightarrow f \rightarrow g_1$, which is infeasible because g_1 should be invoked before g_2 is invoked. In this case we can exclude the chain by giving the constraint $\neg(x^{\mathtt{r}}_{g_2,f} \wedge x^{\mathtt{c}}_{f,g_1})$.

Similarly we exclude a path whose backbone includes a call edge $f \rightarrow g_1$ for the code below, where `input` is the original taint source and f cannot be invoked twice in any taint flow.

```
f(..) { .. g1(..); .. input(..); .. }
```

One can easily see that the path is infeasible and can exclude it by the constraint $\neg x^{\mathtt{c}}_{f,g_1}$.

Also we exclude a path whose backbone includes a return edge g_1 -$\!\!\rightarrow f$ for the code below, where `output` is the final taint sink and f cannot be invoked twice in any taint flow.

```
f(..) { .. output(..); .. g1(..); .. }
```

One can easily see that the path is infeasible and can exclude it by the constraint $\neg x^{\mathtt{r}}_{g_1,f}$.

IV. Example of Finding a Vulnerability

We demonstrate how we efficiently found an integer overflow vulnerability from pngquant-2.7.0 with CVE number [12] using Shovel. The source function of the alarm we classified is `png_read_data`, which reads tainted data from a PNG file with the library function `fread`. The sink function is `rwpng_read_image24_libpng`, which calls `malloc` with `rowbytes * mainprog_ptr->height` as allocation size. Our goal is to determine whether there is taint flow from the taint source to `rowbytes` and `mainprog_ptr->height` in the sink function `rwpng_read_image24_libpng` because such taint flow may cause allocating an overflown size block, which can lead to a security hole.

Step 1 We first obtained a shortest path using Shovel, which is depicted in Figure 5.(A). By browsing the source code with Cscope, we found that `rowbytes` and `mainprog_ptr->height` used in the sink function are defined with `info_ptr->rowbytes` and `info_ptr->height` respectively. Also, we observed `png_set_IHDR` must be visited to define these fields. Therefore we could derive a constraint that taint flow paths should visit the function `png_set_IHDR`.

Step 2 With this additional constraint, we obtained the next path given in Figure 5.(B) from Shovel. By investigating the path using Cscope, we made the following two observations. First, in order to read in tainted input, png_set_IHDR should be called by png_handle_IHDR as in the given call path. Second, `png_handle_IHDR` calls `png_crc_read` to read tainted data into `buf` and this buffer is used to evaluate the argument for `png_set_IHDR` as shown below.

```
png_handle_IHDR(info_ptr, ...){
  char buf[13];
  png_crc_read(buf, ...);
  ...
  height = *(uint*)(buf + 4);
  ...
  png_set_IHDR(info_ptr, ..., height, ...);
}
```

From these observations, we inferred the constraint that `png_crc_read` should return to `png_handle_IHDR` either by the backbone or by a branch.

Step 3 After adding this constraint, we got the shortest path illustrated in Figure 5.(C) using Shovel. We could observe that the return edge from `png_read_end` to the sink function `rwpng_read_image24_libpng` is infeasible. This is because inside `rwpng_read_image24_libpng`, the call to `png_read_end` comes after the `malloc` call. From this we can derive the constraint that the backbone should not include the return from `png_read_end` to `rwpng_read_image24_libpng`.

Step 4 With this additional constraint, Shovel finds the call path given in Figure 5.(D). Finally, this path turned out to carry tainted information from the source to the sink and we could be able to find an exploit and get a CVE number for the vulnerability.

Program	LOC	Func	Edge	Alm	Qry	Res(s)
rand-1.0.4	313	10	11	1	2	0.1
fpgatools-0.0/sort_seq	425	2	1	1	1	0.1
mp3rename-0.6	559	7	6	1	1	0.1
ghostscript-8.71/genconf	1K	12	15	1	1	0.1
vtprint-2.0.2	1K	14	14	1	2	0.1
devio-1.2	1K	37	87	1	1	0.1
bbe-0.2.2	2K	45	88	3	9	0.1
enum-1.1	4K	48	61	1	2	0.1
tiptop-2.2	5K	136	200	4	13	0.1
uni2ascii-4.14	5K	27	26	1	1	0.1
splitvt-1.6.6	6K	121	245	14	28	0.1
pal-0.4.3	7K	120	238	3	4	0.1
rplay-3.3.2	7K	78	174	1	1	0.1
rptp-3.3.2	7K	78	174	5	11	0.1
cdparanoia-3.10.2	12K	222	544	9	24	0.1
shntool-3.0.10	16K	263	755	7	27	0.3
blktrace-1.0.5	17K	148	247	2	4	0.1
dvi2ps-5.1j	20K	593	1572	25	50	0.5
texinfo-6.0/ginfo	23K	505	1707	0(48)	0	0.0
erlang-13.b.3/erl_call	23K	157	305	3	9	0.1
sdop-0.61	23K	166	513	10	38	0.1
zoem-08-248	25K	448	1444	9	36	1.5
latex2rtf-2.3.8	27K	564	2240	9	35	4.6
less-481	27K	453	1255	36	76	0.4
rrdtool-1.4.8	34K	264	698	4	21	0.2
dico-2.0	45K	911	1977	6	44	0.3
dicod-2.0	55K	911	1977	8	36	0.5
daemon-0.6.4	58K	253	543	8	23	0.1
a2ps-4.14	64K	759	1452	22	120	0.6
afbackup-3.5.3	66K	262	799	9	27	0.3
glpk-4.38	95K	1186	4577	6	12	1.1
gnuplot-4.2.6	111K	1815	8627	62(93)	123	5.0
putty-0.65	123K	867	2372	17	66	1.3
TOTAL	927K			290	848	1.3

† The three longest response time spent on generating a path are 25.6, 6.3 and 6.2 seconds.
† The three longest alarm classification processes each required 9, 9 and 8 times of querying to Shovel.

Table I
The overall effectiveness of Shovel in classifying format string vulnerability alarms

V. Experiments

We empirically show the effectiveness of our method on classifying format string vulnerability alarms and integer overflow vulnerability alarms from open source C programs. During this experiment, we were able to find new format string vulnerabilities in 19 programs, and new integer overflow bugs in 5 programs. Two format string vulnerabilities from latex2rtf-2.3.8 and a2ps-4.14 and one integer overflow vulnerability from pngquant-2.7.0 were assigned CVE numbers.

Setting We used Shovel to classify alarms generated from Sparrow [30], [26], a state-of-the-art static analyzer that

Program	LOC	Func	Edge	Alm	Qry	Res(s)
rand-1.0.4	313	10	11	1	1	0.1
enum-1.1	4K	48	61	2	4	0.1
tiptop-2.2	5K	136	200	2	4	0.1
shntool-3.0.10	16K	263	755	19	65	0.3
blktrace-1.0.5	17K	148	247	3	6	0.2
sdop-0.61	23K	166	513	9	14	0.1
libpng-1.6.21/gregbook	41K	224	525	12	36	0.2
pngquant-2.7.0	45K	444	1001	10	46	0.4
dico-2.0	45K	911	1977	3	6	0.1
TOTAL	200K			61	182	0.3

† The three longest response time spent on generating a path are 0.9, 0.7 and 0.5 seconds.
† The three longest alarm classification processes all required 7 times of querying to SHOVEL.

Table II
THE OVERALL EFFECTIVENESS OF SHOVEL IN CLASSIFYING INTEGER OVERFLOW VULNERABILITY ALARMS

aims to verify the absence of fatal bugs in C programs. Since SPARROW is designed based on Abstract Interpretation [7], its analysis is sound in design. The analyzer detects several kinds of errors including format string bugs and integer overflow bugs. Note that other analyzers also can be used because our method is analyzer-independent.

For MinSAT solving, we use the MaxSAT solver Open-WBO [28] with the underlying SAT solver MiniSAT 2.0 [15].

We performed expriments on a Linux 3.10 system using only a single core of Intel Xeon 3.5GHz box with 32GB RAM.

Experimental Evaluation We evaluated our approach on classifying format string vulnerability alarms from 405 open source C programs. The programs were collected from the official UBUNTU package archive from 15 categories (e.g., editors, text processing, network, administration utilities, etc.). SPARROW analyzed the collected benchmark and reported 290 alarms in 33 programs. The LOCs of these programs range from several hundreds to over 100K, and among the 290 alarms 30 alarms are true.

For expreiment on integer overflow vulnerability alarms, we selected a different benchmark since the number of alarms were relatively larger than that of format string vulnerability. From the previously mentioned UBUNTU package, we selected 7 programs whose number of alarms were moderate. And we added 2 more programs (libpng-1.6.21/gregbook and pngquant-2.7.0) related to libpng [27] library, since several interger overflow vulnerabilities were previously found in this library [9], [8]. SPARROW reported 61 alarms from these 9 programs, among which we classified 18 alarms as true.

Table I and II respectively show the expreimental results of classifying format string vulnerability alarms and integer overflow vulnerability alarms. The column labeled **Alarms** shows the number of alarms reported by the SPARROW. The number of functions (**Func**) and the number of call edges (**Edges**) indicate the size of statically estimated call-graphs. **Qry** column shows the total number of times that we queried to SHOVEL during the classification of program's alarms. Lastly, the column labeled **Res** shows the average time spent by SHOVEL for finding the shortest path that satisfies the user constraint.

The experiment shows that the user-interaction based approach can greatly help the user to classify alarms in most of the times. We have successfully classified 351 alarms and identified 48 true bugs (all but one of them were previously unknown bugs) in 25 programs. For most of these alarms, a small number of user feedbacks (2.93 in average, 9 in maximum) were sufficient to determine its trueness. This was possible because we could easily derive a general user constraint from the provided call path. It is true that there were cases where our user-interaction based method does not properly work. For 31 alarms from gnuplot-4.2.6 and 48 alarms from texinfo-6.0/ginfo, we were not able to derive a general condition with the patterns we discussed in Section III.

We observed that SHOVEL is efficient in finding the shortest path that satisfies user constraints. The average response time spent by SHOVEL to find a call path is about 1.3 seconds for format string vulnerability benchmarks and 0.3 seconds for integer overflow vulnerability benchmarks. Our advanced refinement grouping iterated once or more for shntool-3.0.10, latex2rtf-2.3.8. For four alarms from shntool-3.0.10, our algorithm iterated 11, 2, 2 and 2 times. For three alarms from latex2rtf-2.3.8, the algorithm iterated 7, 7 and 1 times. In these cases, the biggest SCC consists of 427 functions out of total 564 functions.

VI. RELATED WORK

CFL-reachability Our work is closely related to context-free language (CFL) reachability problem for program analysis [31], [3], [2], [35]. For a given directed labeled graph and a context-free grammar (CFG), the CFL reachability problem is to find pairs of vertices (u, v) where there exists a path from u to v whose concatenation of labels is in the language of the CFG.

It is easy to see that our notion of well-formedness can be defined by a CFG. In the context of CFL reachability, our goal can be viewed as finding the shortest CFL reachable path from source to sink satisfying some constraints on edges (*i.e.,* user's constraints). To the best of our knowledge, we know of no algorithm for solving the *constrained* shortest CFL reachability problem. Reps et al. [31] proposed a polynomial algorithm for the CFL reachability problem, which however does not find a shortest one. Osbert et al. [3] proposed a polynomial algorithm for finding a shortest CFL reachable path, but the algorithm does not consider any

constraint.

SMT for graph properties There are prior works that use SMT solvers for finding graphs satisfying constraints such as reachability, shortest-paths, minimum spanning tree [4], [17]. However, these approaches cannot handle reachability constraints for *cyclic* graphs, whereas our work can handle them (we even handled SCCs with 400 nodes).

Answer Set Programming Unlike SAT, Answer Set Programming (ASP) [1], [18], [19], [29], [33] can encode reachability constraints in cyclic graphs. Boolean constraints on edges also can be encoded. The resulting logic encoding is always both sound and complete, but checking satisfiability/-validity of ASP formulas is computationally more expensive than checking propositional formulas (SAT) in general. For example, Agostino et al. [14] conducted experiments using two SAT-based AST solvers, called SMODELS [29] and CMODELS [19], for the hamiltonian path problem. For a relatively small-sized graph instance (named 2xp30.4 in the paper, #nodes = 60, #edges = 318), SMODELS exceeded 180 minutes time limit, and CMODELS spent about an hour and half.

Inlining-based Approaches A simple and immediate solution to our problem is simply removing calls and returns by inlining all function calls a fixed number of times as in [20], [6], [34]. Thus, it has one kind of edges between nodes, so that they can avoid encoding the constraint of matched calls and returns.

But this simple solution is not applicable for two reasons: potentially exponential-size encoding, and unsoundness for programs with recursive function calls. Our method is both effective and sound in the presence of cycles in call-graphs.

Supporting Manual Inspection of Alarms The work by Zhu et al [36] is most closely related to our user-interactive approach to information flow analysis alarms. When source code of libraries is missing, their technique infers a smallest set of must-not-flow requirements on library functions that are sufficient to ensure that a given program is free of source-sink errors via abductive inference [13]. Their system contains a refinement loop where partially confirmed inferred specification will lead to another minimal specifications.

Their approach is similar to our work in the sense that both approaches aim to filter out false positives minimizing user interactions. But their method does not aim to display feasible paths as ours. So it is not helpful for understanding and finding real errors.

Osbert et al. [2] automated sanitizer placement for preventing runtime information flow errors. They also aim to minimize user-interaction to find appropriate instrumentation points. But also, their method is not helpful in understanding true alarms.

In addition, our work can be combined with other techniques for a more sophisticated interface to reduce alarm investigation efforts. Non-statistical clustering techniques group alarms of the same root causes [25], [26]. Those techniques can be used to reduce the number of alarms to inspect. Semantic slicing using abstract dependences [30], [32] can be used to highlight only relevant parts of resulting call-graphs.

Supplementary Material The soundness proof, the crash bugs and the details of user interaction in our experiment are all available at the project website: http://sf.snu.ac.kr/shovel

VII. Appendix

Definition of Advanced Initial Groupings

$$\begin{aligned}\mathsf{InitGrps}(\mu) \stackrel{\text{def}}{=}\ & \{\,\{v\} \mid v \in G\,\} \\ & \cup \mathrm{SCC}(R(\mu)) \cup \mathrm{SCC}(C(\mu)) \cup \mathrm{SCC}(B(\mu))\end{aligned}$$

where

$$\begin{aligned}R(\mu) &\stackrel{\text{def}}{=} \{\, v \dashrightarrow w \in G \,\} \setminus \{\, v \dashrightarrow w \mid \mu \Rightarrow \neg x^{\mathsf{r}}_{v,w} \,\} \\ C(\mu) &\stackrel{\text{def}}{=} \{\, v \to w \in G \,\} \setminus \{\, v \to w \mid \mu \Rightarrow \neg x^{\mathsf{c}}_{v,w} \,\} \\ B(\mu) &\stackrel{\text{def}}{=} \{\, v \to w \in G \,\} \setminus \{\, v \to w \mid \mu \Rightarrow \neg x^{\mathsf{b}}_{v,w} \,\}\end{aligned}$$

Definition of Advanced Refinement Groupings

$$\begin{aligned}&\mathsf{RefiGrps}(p) \stackrel{\text{def}}{=} \\ &\quad \mathrm{SCC}(R(p)) \cup \mathrm{SCC}(C(p)) \cup \mathrm{SCC}(B(p)) \\ &\cup \{\, V \cup \overleftrightarrow{\mathrm{NB}}(V, \mathrm{SrcTgt}(R(p) \cup C(p))) \mid \\ &\qquad\qquad V \in \mathrm{SCC}(R(p)) \cup \mathrm{SCC}(C(p)) \,\} \\ &\cup \{\, V \cup \overrightarrow{\mathrm{NB}}(V, \mathrm{SrcTgt}(R(p) \cup C(p) \cup B(p))) \mid \\ &\qquad\qquad V \in \mathrm{SCC}(B(p)) \vee V = B(p) \,\}\end{aligned}$$

where

$$\begin{aligned}R(p) &\stackrel{\text{def}}{=} \{\, v \dashrightarrow w \mid x^{\mathsf{r}}_{v,w} \in p \,\} \\ C(p) &\stackrel{\text{def}}{=} \{\, v \to w \mid x^{\mathsf{c}}_{v,w} \in p \,\} \\ B(p) &\stackrel{\text{def}}{=} \{\, v \to w \mid x^{\mathsf{b}}_{v,w} \in p \,\} \\ \mathrm{SrcTgt}(X) &\stackrel{\text{def}}{=} \{\, v, w \mid v \to w \in X \vee v \dashrightarrow w \in X \,\} \\ \overleftrightarrow{\mathrm{NB}}(V, W) &\stackrel{\text{def}}{=} \{\, v \notin W \mid (v \to w \text{ or } v \dashrightarrow w) \in G \wedge w \in V \,\} \\ \overrightarrow{\mathrm{NB}}(V, W) &\stackrel{\text{def}}{=} \{\, v \notin W \mid v \to w \in G \wedge w \in V \,\}\end{aligned}$$

References

[1] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.

[2] Osbert Bastani, Saswat Anand, and Alex Aiken. Interactively verifying absence of explicit information flows in android apps. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 299–315, New York, NY, USA, 2015. ACM.

[3] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 553–566, New York, NY, USA, 2015. ACM.

[4] Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. Sat modulo monotonic theories. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 3702–3709. AAAI Press, 2015.

[5] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[6] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[8] Cve-2013-7353. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-7353.

[9] Cve-2013-7354. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-7354.

[10] Cve-2015-8106. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8106.

[11] Cve-2015-8107. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8107.

[12] Cve-2016-5735. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5735.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 181–192, New York, NY, USA, 2012. ACM.

[14] Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *J. Exp. Theor. Artif. Intell.*, 21(2):79–121, June 2009.

[15] Niklas Een, Alan Mishchenko, and Niklas Sorensson. Applying logic synthesis for speeding up sat. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, SAT'07, pages 272–286, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[17] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. Sat modulo graphs: Acyclicity. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence - Volume 8761*, pages 137–151, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[18] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Clasp: A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR'07, pages 260–265, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

[20] William R. Harris, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Program analysis via satisfiability modulo path programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 71–82, New York, NY, USA, 2010. ACM.

[21] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[22] Yungbum Jung and Kwangkeun Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 131–140, New York, NY, USA, 2008. ACM.

[23] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In Hao Chen, Larry Koved, and Dan S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*. IEEE, May 2012.

[24] L. Layman, L. Williams, and R. S. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 176–185, Sept 2007.

[25] Wei Le and Mary Lou Soffa. Path-based fault correlations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 307–316, New York, NY, USA, 2010. ACM.

[26] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. In *Proceedings of the 13th International Conference on Verification, ModChecking, and Abstract Interpretation*, VMCAI'12, pages 299–314, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] Libpng. http://www.libpng.org/.

[28] Ruben Martins, Vasco Manquinho, and Ins Lynce. Open-wbo: A modular maxsat solver,. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing, SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer International Publishing, 2014.

[29] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A system for answer set programming. *CoRR*, cs.AI/0003033, 2000.

[30] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for c-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 229–238, New York, NY, USA, 2012. ACM.

[31] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[32] Xavier Rival. Abstract dependences for alarm diagnosis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 347–363, Berlin, Heidelberg, 2005. Springer-Verlag.

[33] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, Sept 2011.

[34] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[35] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.

[36] Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 290–306, New York, NY, USA, 2013. Springer-Verlag New York, Inc.